Pascal
NEWSLETTER

NUMBER 6 OREGON SOFTWARE SPRING 1983

Pascal-2 Version 2.1 to be released

Oregon Software now has Version 2.1 of our optimizing Pascal compiler in field test for RSX, RSTS, and RT-11 systems. Version 2.1 for all systems will be released to general users June 1, beginning with RSX.

The upgrade from V2.0 represents a major revision of the Pascal–2 software, to include conformant array parameters, improve error diagnostics, enhance performance, remove size limitations on user programs, and fix all major bugs collected in the last year.

Immediately after field test, Version 2.1 will be released to all customers who have reported bugs since Version 2.0K was released last year. These shipments will probably be concluded before June 1. Then, the V2.1 update will be released to all supported customers who send us a written request.

With the energy that has gone into producing V2.1, we did not release any interim updates after V2.0K. For this reason, we will send Version 2.1A to any customer who renewed support after the release of V2.0K, even if that customer's support expires before the release of V2.1A. Updates are free on floppies and magtape; a fee is charged for other media. All customers must request V2.1 in writing.

Because many new features have been added and because the code generated by the V2.1 compiler is different from that of V2.0, users will need to modify and recompile existing Pascal–2 programs once they have installed V2.1. Users may wish to redesign existing programs to take advantage of the new features such as conformant array parameters, lazy I/O and user control of run-time error reporting.

With the addition of conformant array parameters, the Pascal–2 compiler conforms to Level 1 of the proposed ISO standard (ISO dp7185). With conformant array parameters, users can write general procedures that accept arrays with different lower and upper bounds. With V2.0, users have to write a new procedure for each array that varies in size or bounds.

We have substantially improved the error diagnostics by including a procedure walkback generated as a result of a run-time error. The walkback displays the error message, the point of error in Pascal source terms, and a traceback of procedure calls leading to the error. Version V2.0 has no equivalent run-time diagnostics.

In addition, a new I/O error trapping capability allows users to recover from I/O errors with their own code, to change the wording of run-time error messages and to print additional information besides that printed in the walkback.

Other substantive changes include: the addition of "lazy I/O," a scheme by which data is not read or written until actually used in the program; elimination of certain size restrictions, allowing larger programs to be compiled; capability to include more procedures and type definitions; new procedures to do common and useful I/O procedures, including delete and rename; and features to help users of small systems.

Changes and additions to V2.1 are documented in a 70-page supplement to the *Pascal–2 User Manual*. The supplement will be supplied with all updates. The changes will be included in a new edition of the manual to be printed in the late spring.

This newsletter carries a summary of the major changes, plus articles on the use of various new features. Major bug fixes are described in the Bug Log.

In this issue...

UNIX version joins Pascal-2 family

Oregon Software has released the Pascal-2 compiler on the UNIX operating system for the PDP-11. The new product includes an enhanced version of our interactive source-level debugger.

UNIX, originally developed by Bell Laboratories, is sold through license agreements with Western Electric. Popular in university environments originally, UNIX is becoming a major force in software development applications and has spawned many UNIX "look-alike" operating systems.

A key feature of UNIX is the collection of supporting "tools" that provide a software developer with standard, easy-to-use programs to help with coding and development. Oregon Software's Pascal compiler system contains similar kinds of tools — debugger, execution profiler, and development utilities.

The debugger allows the programmer to work interactively at the source level to solve logic errors in applications, thus simplifying and speeding development. The debugger runs as a separate process from the application code that is being debugged, keeps track of multiple compilation units, and performs breakpoint debugging without slowing down the application code.

The profiler aids the applications programmer by identifying execution bottlenecks. With data supplied by the profiler, the developer can reorganize portions of the code to substantially improve overall execution speed.

The Oregon Software Pascal-2 UNIX compiler supports all capabilities of standard Pascal and conforms to the draft proposed Pascal standard (International Standards Organization dp7185.1, level 0). As a result, users are assured that code developed under Pascal-2 can be moved to other systems with standard Pascal compilers.

Pascal-2 for UNIX is available now directly from Oregon Software or from authorized distributors.

SourceTools available for RSX

Oregon Software is now releasing SourceTools for RSX, including VMS in compatibility mode, to general end users. The seven-program package helps programmers develop and coordinate large software projects. SourceTools has been in field test for the last three months and no major problems have been reported.

The source-control system, SourceCon, is the foundation of the package. SourceCon monitors files placed under its control, to prevent programmers from making simultaneous and conflicting changes and to keep a development history of the source files. In a sense, SourceCon provides a "library service" for programmers and writers, recording changes to text files and the reasons for them, and ensuring that earlier versions of a module are always available.

Another program in the SourceTools package simplifies and automates the task of rebuilding software from component parts. The SourceTools MAKE program automatically determines which files are out of date, then executes only the commands needed to rebuild those out of date files.

In a more conventional environment, the programmer must remember how all the components of a large program fit together whenever some components are modified, and must recall exactly how to re-create each component so that it fits with others in the module. This is the old "recompile everything in sight" problem. Or, the programmer has to keep a set of command files around for each of the various parts and for combining the parts together.

Two additional utilities alleviate the tedium of maintaining parallel or alternate versions of files. A text comparison program, TEXTCOM, isolates differences between two text files. A stream editor, SEDIT, interprets a text-editor script and applies the editor-commands to its input text file. By a switch in the text comparison program, TEXTCOM can output an editor script compatible with SEDIT. This combination can be a powerful tool for maintaining parallel files on remote systems and for cross-development work.

These time-saving software tools are used extensively by our own programmers.

Summary of changes and new features for Pascal-2 Version 2.1

New features provide the following benefits for users:

- Conformant array parameters allow you to write general procedures that accept array parameters of different sizes and with different lower and upper bounds.
- Non-decimal constants allow you to use constants in number bases ranging from base 2 (binary) to base 16 (hexadecimal).
- Run-time error walkback aids in diagnosis of run-time errors.
- User-processing of run-time errors gives you control of run-time error reporting. You can change the wording of run-time error messages and print additional information besides the walkback. The walkback may be disabled with the **nowalkback** switch.
- I/O error trapping capability allows you to recover from I/O errors with your own code. Article on page **6** .
- A new procedure provides an explanation of I/O errors (RSTS and RSX only). Article on page **6** .
- Lazy I/O. A new I/O scheme in which data is not read or written until actually used in the program. Article on page **10** .
- New I/O control switches extend the capabilities of the file system. See related article on page **10** .
- New procedures **getpos** and **setpos** simulate random access to files of type **text**. Article on page **8** .
- New syntax of **%include** directive allows you to specify the disk volume number, UIC and version number of the included file.
- New built-in procedures **rename** and **delete** allow you to rename or delete files from within the program.
- New function **space** determines the amount of stack and heap available to an executing program.
- Eight-bit characters allow you to use extended character sets.
- **WK:** specification allows you to specify the device to which the compiler directs its working data files and helps small-systems users reduce load on the default disk.

The following changes have been made:

- Certain size restrictions have been eliminated to allow larger programs to be compiled. Also, V2.1 programs can have more procedures and more type definitions than V2.0 programs.
- New run-time error numbers and messages have been added.
- Single-character I/O has been added to RSX. Article on page **4** .
- New method of reading MCR command lines has been added to RSX.
- New ways to overlay Pascal–2 programs; new way to overlay the Pascal–2 support library.
- New support library interface. Library entry points have been changed to the form **P$***nnn*, where *nnn* is a number in the range 0..135.
- Procedure **timestamp** added to RSX. Returns the date and time.

Pascal-1 V1.3 in field test

Version 1.3 of Pascal–1 is also in field-test for RSX, RSTS, and RT-11 systems. V1.3 also will be made generally available to end users by June, along with Pascal–2 V2.1.

V1.3 will include lazy I/O, a new library interface, and a new initialization interface. Pascal–1 users will be able to use the new I/O error diagnostics, but they must change their source programs.

A number of bugs also have been fixed.

Terminal I/O for RSX

For each operating system, the terminal driver acts as an intermediary between a Pascal program and the terminal that is running that program. The terminal driver is record-oriented, which means it sends a full line of text to the screen or to the program instead of sending one character at a time.

For example, when a **read** statement reads input from a terminal, the terminal driver reads each character and places it in an internal buffer until the terminal driver encounters an end of line (a carriage return). At this point, the terminal driver sends the entire line to the program and lets the program process the line one character at a time.

The same is true for **write** statements, except that the terminal driver buffers the characters being sent to the terminal until a **writeln** clears the buffer or until the buffer size is reached. Then the record is displayed on the screen. The size of the buffer can be controlled with the **buffersize**:*n* file control switch. See "Single-Character I/O for RSX."

As mentioned above, the **writeln** statement instructs the terminal driver to send the output to the screen. After the terminal driver prints a line, it positions the cursor over the first character of the line it just printed. The usual output sequence sent by the operating system's terminal driver is: line feed, data, carriage return. This sequence moves the cursor to the next line, prints the data and returns the cursor to the first position of the newly printed line. This method of writing lines to a terminal is different from other operating systems where the normal output sequence is data, carriage return, and line feed (print the record, move to the first character of the line, then move to the next line).

The normal sequence of commands issued by the terminal driver may not particularly suit special I/O applications such as direct cursor addressing. On RSX, for example, the line-feed and carriage-return characters are often not needed or are sometimes unwanted. To gain control of the terminal-I/O command sequence, use the **ftn** file control switch , a feature which allows you to override the effects of the terminal driver (see "Using FORTRAN Carriage Control under Version 2.1 for RSX").

Single-character I/O added for RSX

The **buffersize**:*n* I/O control switch sets the line size of a **text** file to *n* bytes. By setting your terminal's internal line size to 1 with **buffersize**:1, you can write programs that perform single-character input and output. This feature is useful when you need to do special output formatting to a video terminal, such as direct cursor addressing of output on the screen.

CAUTION

Single-character input/output increases system overhead and should be used with care. Overhead is increased because the monitor must be called for each character read from and written to the terminal. Several users simultaneously outputting in single-character mode can significantly reduce the system response time.

To enter single-character mode, specify the **buffersize**:1 switch on the **reset** statement for single-character input and on the **rewrite** statement for single-character output. In single-character mode, the terminal driver reads and writes the characters as usual. But since the internal buffer size is one byte, each new character fills the buffer, causing the buffer to be emptied. In other words, each character written via **write** is immediately printed on the screen; each character that you enter is immediately sent to the program.

Continued on Page 8

```
program Single;
  var
    Ch: char;

  begin
    reset(input, 'TI:/buff:1');
    rewrite(output, 'TI:/buff:1');
    write('Type a message: ');
    while not eoln do
      begin
        read(Ch);
        write(Ch);
      end;
  end.

>RUN SINGLE
Type a message: ddoouubbllee_ yviissiioonn
>
```

Program Single uses a combination of single-character input and single-character output procedures under the RSX operating system. When compiled and linked and run, the program simply echoes each entered character as shown.

Using FORTRAN carriage control under Version 2.1 for RSX

Pascal–2 allows your Pascal programs to write **text** files that follow the FORTRAN standard output conventions. To do this, specify the **ftn** file control switch on the **reset** or **rewrite** statement that opens the file. For interactive I/O, the **ftn** switch is used with the standard file **output**.

FORTRAN conventions state that the first character of each line of an ASCII file is nonprintable and is used to control the vertical formatting of an output file or terminal screen (see table of commonly used vertical formatting characters). Most characters have no useful meaning in the first position; however, some characters significantly affect the format of the output. The *RSX-11M/M-Plus I/O Drivers Reference Manual* contains a complete list of these vertical formatting characters.

The null character (**chr(0)**) can be used to prevent the terminal driver from adding any special characters to the data you write to the terminal. You will have to insert the carriage-control characters yourself, as shown in the last line of sample program **Ftn**. In the output shown for program **Ftn**, the "overprint" **writeln** replaces the characters "Normal output" from the previous **writeln** with "Overprint****," resulting in a different line of text. If you set your terminal to a slow baud rate, you would actually see the overprinting occur. If you direct the file to a printer, the overprinted line will contain overstrikes.

The **ftn** switch does not solve all terminal I/O problems. The data is not written to the terminal immediately; the characters are still buffered until a **writeln** is executed. You can overcome this problem by using the feature that allows single-character I/O.

```
program Ftn;

  begin
    rewrite(output, 'TI:/ftn');
    writeln(' Normal output');
    writeln(' More normal output');
    writeln('0Double space');
    writeln('1Page eject');
    writeln(' Normal output of a long line');
    writeln('+Overprint****');
    writeln('$Prompt output: ');
    writeln(chr(0), 'Internal format',
            chr(13), chr(10));
  end.


>RUN FTN
Normal output
More normal output

Double space

<ff> ———————— skips to top of next page

Page eject
Overprint**** of a long line
Prompt output: Internal format

>
```

Program Ftn shows the use of FORTRAN standard output conventions in a Pascal program under RSX. The first character written on each line is interpreted as the vertical-format control character. **Chr(13)** is the carriage-return character, and **chr(10)** is the line-feed character. The output shown appears on your terminal when the program is run.

Commonly Used Vertical Formatting Characters

Character	Meaning	Output Sequence
<space>	Normal output	Line feed, data, carriage return
0	Double-space	Two line feeds, data, carriage return
1	Page eject	Form feed, data, carriage return
+	Overprint	Data, carriage return
$	Prompt	Line feed, data, remain on same line
<null>	No special formatting	Data only

New routines trap I/O errors

Pascal–2 V2.1 permits you to write programs that trap and detect many kinds of I/O-related (normally fatal) errors. Using three predefined routines to process I/O errors with your own code, you can terminate the program at the occurrence of an I/O error or continue execution in spite of the error. You can print your own diagnostics with a fourth procedure.

The three error-trapping routines — procedure **noioerror** and functions **ioerror** and **iostatus** — are predefined and do not need to be declared in your program. They accept a file variable as their only parameter. The **sayerr** procedure, which prints the text of a given error messge, is not predeclared, so you must declare it when used. Although details vary slightly according to the operating system, these routines routines work as follows.

Procedure **noioerror** specifies that the calling program will handle any I/O errors that result from reading or writing to the specified file. The file must be open before **noioerror** is called.

Function **ioerror** determines the status of the last I/O operation that the program performed on the specified file. This **boolean** function returns a **true** value if an I/O error has occurred or a **false** value if the operation was successful.

Function **iostatus** helps your program determine the cause of the error by returning an integer error code describing the last attempt to access a file. Your program can either bypass the problem and continue processing, or terminate so you can correct the problem.

You can pass **iostatus** as a parameter to **sayerr**, an **external** procedure which prints the text of the error message corresponding to the value returned by **iostatus**.

Continued on Page 7

Printing I/O errors under RSX

Procedure **sayerr** prints the text of a given error message. A negative error code indicates that the error is specific to the RSX operating system. A positive error code indicates that the error is detected by the Pascal–2 support library. Pascal–2 error codes, along with the text of the error message and a brief explanation of the cause, are listed in the V2.1 supplement to the *Pascal–2 User Manual*. RSX I/O error codes are listed in the *RSX I/O Operations Reference Manual*.

As an external procedure supplied in the Pascal support library, you must declare the **sayerr** procedure in your program as shown:

```
procedure Sayerr(Status: integer);
  external;
```

where **Status** is the error code of the error.

Procedure **sayerr** takes an RSX I/O error code (a negative number) and looks in the file LB:[1,2]QIOSYM.MSG to print the text of the error message. **Sayerr** only prints messages for negative I/O error codes in the range −255.. − 1; **sayerr** ignores error codes that lie outside this range, printing nothing.

```
program Opnerr;
  var
    F: text;
    Status: integer;

  procedure SayErr(Code: integer);
    external;
  begin
    reset(F, 'XXXX:', 'test.dat', Status);
    if Ioerror(F) then
      begin
        writeln('I/O status=', Iostatus(F));
        SayErr(Iostatus(F));
      end;
  end.
>RUN OPNERR
I/O status=    -55
Bad device name ———— message sayerr prints
>
```

Program Opnerr shows the use of procedure **sayerr** on RSX. The program attempts to open a file called TEST.DAT on a ficticious device with the name XXXX:. Normally, this program would fail with no specific indication of what caused the **reset** failure. The error is detected by **Ioerror**. Function **Iostatus** returns the value -55 and passes the parameter to **sayerr**. The call to **sayerr** prints the text of the message for RSX I/O error code -55: "bad device name." When this program is compiled and executed on RSX, it yields output similar to that shown above. A RSTS example produces similar results.

```
program Iotest;

  var
    I,Times: integer;

  begin
    Noioerror(input);
    for Times := 1 to 5 do
      begin
        write('Type an integer: ');
        read(I);
        if Ioerror(input)
          then writeln('Error detected.  Status=', Iostatus(input))
        else writeln('The integer was: ', I: 1);
        readln;
        writeln;
      end;
  end.
```

```
>RUN IOTEST
Type an integer: 1234
The integer was: 1234

Type an integer: 123456789 ———————————————— integer too large
Error detected.  Status=    19 ———————————— Pascal-2 error code for invalid integer

Type an integer: ffg ———————————————————— non-integer characters
Error detected.  Status=    19 ———————————— Pascal-2 error code for invalid integer

Type an integer: 77
The integer was: 77

>
```

Program Iotest illustrates the use of the use of pre-defined error-trapping routines for RSX. This program continues to execute after an I/O error is found; without these routines, the first error would cause the program to abort. The call to **noioerror** notifies the run-time system that the program will handle errors detected on the standard file **input**. When compiled and run, the results of such a program are similar to the above. The first entry results in a successful read of the integer **I**. The second and third entries result in a Pascal-2 run-time error. The final entry is successfully read, and the program ends. RSTS and RT examples produce similar results.

When you call these routines, you are responsible for checking the status of each I/O operation, to ensure that it was successful. If you fail to check the status afterwards, the results will be unpredictable.

The I/O error-trapping procedures can be used to determine the reason that a file could not be opened. To use this feature, specify the fourth parameter on calls to **reset** and **rewrite**. The use of the fourth parameter keeps the **reset** or **rewrite** from trapping a normally fatal "open" error. This allows your program to recover and continue, or terminate under your control. (Sample program OPNERR demonstrates use of this feature under RSX.)

Random access to text files simulated

Because lines of text vary in length, the **seek** procedure cannot predict the location of a line within a text file. The Pascal-2 support library for V2.1 supplies two external procedures, **getpos** and **setpos**, that simulate random access to **text** files. Working together, these procedures use a set of values to locate the next line of text. **Getpos** determines the starting location of the line and **setpos** returns the file pointer to the specified starting location of a line within the file.

For text files, the beginning of each line is denoted by a block number and a byte offset into that block. Each block contains 512 bytes. The first line of a file starts with block 1, offset 0. If you try to access a nonexistent position or a position in the middle of a line, an I/O error will result. When the file is read, use **getpos** to determine the starting position of the next line, and save that block and offset combination for later use by **setpos**. The block number and byte offset must be values returned by **getpos**. You cannot compute the values yourself.

These two procedures are not predefined and must be declared in your program as **external**. These procedures do not provide "true" random access; you cannot use them to access a file you have not previously read.

Sample program REVERSE shows the use of **getpos** and **setpos**.

'Getpos' procedure

Procedure **getpos** determines the starting position of the next line to be read from or written to a text file. **Getpos** requires three parameters, which are passed by reference, as shown below. Together, the parameters **Block** and **Offset** point to the next line to be processed.

```
procedure GetPos(var F: text;
    var Block, Offset: integer);
    external;
```

where

F	is the file variable of type **text**.
Block	is the returned disk block number of the next line in file F to be read or written.
Offset	is the returned byte offset into **Block**.

You should always call **getpos** to obtain the location in the file before you call **setpos**, so the block and offset values being passed to **setpos** are valid.

'Setpos' procedure

Procedure **setpos** positions the file pointer to a specified block number and byte offset into that block. **Setpos** accepts the same three parameters as **getpos**, except **Block** and **Offset** are passed by value. The **setpos** declaration is:

```
procedure SetPos(var F: text;
    Block, Offset: integer);
    external;
```

where

F	is the file variable of type **text**.
Block	is the block number to which the file pointer is set.
Offset	is the byte offset into **Block**.

Together, **Block** and **Offset** point to the new position. To stress an earlier point, the block number and byte offset must be values returned by **getpos**. Do not attempt to compute the values yourself. Save the returned values for later use.

Errors

If an error is detected while **setpos** tries to position the file, the end-of-file flag **eof** is set to true. The **ioerror** and **iostatus** support library procedures may help you to determine the reason that the line could not be accessed. (For details on **ioerror** and **iostatus** see article on page _ .) If a file is positioned to a block and offset that does not correspond to the first character of a line, the results will be unpredictable.

Using FORTRAN... Continued from Page 4

No special formatting characters are inserted when you use the **write** statement (see "Using FORTRAN Carriage Control under Version 2.1 for RSX"). A **writeln** statement will print a carriage return followed by a line feed, as if the buffer were empty. You can supply formatting characters by using the **chr** function to generate the appropriate characters.

For single-character input, you do not need to type a carriage return after each character to signify the end of the line.

```
program Reverse;

{ Print the contents of a file in reverse line order }
  type
    Pointer = ^position;
    position =
      record
        Next: pointer;
        Block: integer;
        Offset: integer;
      end;
  var
    F: text;
    Filename: packed array [1..80] of char;
    P, X: pointer;

  procedure GetPos(var F: text; var Block, Offset: integer);
    external;

  procedure SetPos(var F: text; Block, Offset: integer);
    external;
  begin
    write('File name? ');
    readln(Filename);
    reset(F, Filename);
    P := nil;
    repeat                          { read the file }
      new(X);
      with X^ do GetPos(F, Block, Offset);  ---------- get start of next line
      X^.Next := P;
      P := X;
      readln(F);
    until eof(F);

    while P <> nil do               { write the file }
      with P^ do
        begin
          SetPos(F, Block, Offset);  ---------- set pointer to start of next line
          if not eof(F) then
            begin
              while not eoln(F) do
                begin
                  write(F^);
                  get(F);
                end;
              writeln;
            end;
          P := P^.Next;
        end;
  end.
```

Program Reverse demonstrates the use of the **getpos** and **setpos** procedures to simulate random access of text files on RSX, RSTS, and RT. The program reads a text file and saves the position of each line in a linked list. It then prints the file in reverse line order so that the last line of the file is printed first, the next-to-last line is printed second, and so on, with the first line printed last.

Lazy I/O interface for all systems

For standard Pascal, an interactive input file, such as a terminal, poses a problem. A program must always be able to determine the current status of an open file, i.e., it must be able to retrieve the current record from the buffer variable (`F^`) and to check the current values of **eoln** and **eof**. Since an interactive file is being created as it is being read, the program must periodically wait for additional input to determine the file's current status. With the Pascal support library's new approach to interactive I/O, the program should not wait for input at inconvenient times.

Pascal–2 Version 2.1 uses an input interface known as "lazy I/O" to handle input from **text** files. Since a file's status needs to be defined only when the program actually refers to it, lazy I/O safely delays any input operation until the program uses its results. When a Pascal program requests an input operation on a text file, the operation is recorded for later use. The delayed operation is triggered by any subsequent reference to the file's buffer variable, **eof** value, or **eoln** value. The delay is invisible to the program but is visible to the user from the way the program is synchronized with interactive input.

To use lazy I/O, you need to be aware of its effect on synchronization of input and output operations. As an example, consider a simple program that reads its standard file **input**, which is connected to a terminal. The program prompts for each line and stops at the end of the file. The design of the program is dictated by two requirements:

- For the prompt to be effective, it must appear before the user is required to type the line.
- To detect the end of the file correctly, the program must check for it before reading each line.

To meet both of these requirements, the program must print the prompt before performing any operation that requires the next line of the file to be known: e.g., checking for "end of file" or reading the line. The sample programs in the "Conversion Note" show the differences in design between Version 2.0 and 2.1.

CONVERSION NOTE

Lazy I/O changes the compiler's implementation of interactive I/O. Programs compiled with Pascal–2 Version 2.0 may have to be revised to conform with the new schema. For example, under V2.0 program INTERACTIVE is coded as follows.

```
program Interactive(input, output);
{sample for version 2.0}

  begin
    while not eof do
    begin
     write('prompt:');
     readln;
    end
  end.
```

The program above compiles and executes under Pascal–2 V2.1, but the program is not synchronized with input from the terminal. As a result, you do not see the prompt until you type in a line. In other words, you are prompted for the line you have just entered. For the prompt to be effective, program INTERACTIVE should be coded as follows.

```
program Interactive(input, output);
{sample for version 2.1}

begin
  write('prompt:');
  while not eof do
  begin
    readln;
    write('prompt:');
  end
end.
```

NOTE: Under the Digital operating systems, typing Control-Z (`^Z`) in response to the prompt signals an "end of file" on the interactive input file, halting the program.

Converting unsigned integers on the PDP-11

PDP-11 computers store integer variables in 16-bit words. These 16-bit words may be interpreted as signed or unsigned integers. As a signed number, in two's complement notation, 16 bits can represent numbers in the range of −32767..32767. When considered as unsigned numbers, a word can represent an integer in the range of 0..65535. For both signed and unsigned integers, the bit patterns in the word are the same; only the interpretation of the bit patterns differs.

When their values are compared or used in mathematical expressions, unsigned integers differ greatly from signed integers. As an example, consider a word in which all 16 bits are set to one. This word has a value of -1 when interpreted as a signed integer, or a value of 65535 interpreted as an unsigned integer. When this word is compared with some other value, the PDP-11 uses different combinations of instructions for signed and unsigned comparisons. If this number is multiplied by two, the result is a value of -2 for signed or 131070 for unsigned. The latter is an overflow condition because the result will not fit within 16 bits.

The Pascal-2 compiler and support library also differ in their treatment of signed and unsigned integers. The compiler can deal with unsigned integers, but the library contains a single routine that interprets all integers as signed values. For example, if you define a variable to be of type **integer** in your Pascal program, the compiler treats that value as a signed integer, unless you specify an unsigned integer using a subrange notation such as:

```
type
  unsigned = 0..65535;

var
  X: unsigned;
```

According to your data declarations, the compiler generates the correct code to compare, multiply, or divide unsigned numbers. However, the support library only prints out a signed integer unless you use the following procedure in your program instead of the **write** statement:

```
procedure Uwrite(X: unsigned;
                 Width: integer);

  var
    k: integer;

  begin
    if (X > 32767) and (Width >= 0) then
      begin
      if Width > 0 then
        Width := Width - 1;
        write(X div 10: Width);
        X := X mod 10;
        Width := 1;
      end;
    write(X: width);
  end;
```

This procedure takes an unsigned integer and a field width as its parameters. The number is printed as a value in the range of 0..65535, right justified in the specified field.

The PDP-11 floating-point hardware uses a signed conversion when it converts an integer value to a real value. If you wish to convert an unsigned integer to a real number, you should use the following function.

```
function Ufloat(X: unsigned): real;

  var
    R: real;

  begin
    R := X;
    if R < 0.0 then
      R := R + 65536.0;
    Ufloat := R;
  end;
```

This function takes an unsigned integer as its parameter and returns a real value in the range of 0.0 to 65535.0. The **trunc** and **round** procedures convert real numbers to integers. The floating-point hardware assumes a signed conversion, so the following function should be used when an unsigned integer result is desired:

```
function Utrunc(R: real): unsigned;

  begin
    if (R > 65535.0) or (R < 0.0) then
      writeln('Unsigned number out of range');
    if R > 32767.0 then
      R := R - 65536.0;
    Utrunc := trunc(R);
  end;
```

This function takes a real number in the range 0.0 to 65535.0 and converts it to an unsigned integer. The unsigned **round** function is very similar to the above unsigned **trunc** function.

The Log: Pascal-1 and Pascal-2

Pascal–2 V2.1 fixes a number of bugs, including many that generated obscure or random behaviors. Several fixes listed for V2.1 involve the support library; such fixes also apply to V1.3.

Pascal-2

V2.1 fixed the bugs consistently generating these symptoms:

/EIS, /FIS Arithmetic

Real operations under `/eis` and `/fis` generated incorrect results at times. Problem occurred in the handling of intermediate, stacked results of the simulated real arithmetic (fis instructions) operations.

Packed Records

Packed records would generate incorrect results if the record was moved to the odd byte of the word.

Odd Address Traps

Several problems, resulting in odd-address traps at run time, were fixed.

Odd Address Traps During Set Operations

Compiler assumed that sets were always word-aligned (true unless the set is only one byte long). And it didn't ensure that the two operands were unpacked.

Illegal use of MOD

Illegal use of `mod` caused a compiler trap. An error message is now generated.

'%Include' Syntax Checking

Previously, the compiler didn't catch errors in syntax. It now generates the proper error message.

'Repeat .. Until' Errors

Errors were occasionally generated in `repeat .. until` operations.

Expression Evaluation Errors

During optimization, the compiler would sometimes evaluate an expression when it was not safe to do so.

Boolean Assignments in Debugger

If the first Debugger command in a program being debugged involved assigning a value to a boolean variable, the results would be "out of range."

Debugger 'L' Command

The Debugger `L` command was not ignoring enough lines at the top of each listing page; part of the page header was being included in the listing of the code.

Debugger, File Pointers

The Debugger had problems with pointers to files, either printing the wrong value or trapping.

Debugger, Packed Records

The Debugger was not making the transition between unpacked and packed parts of a record properly. Incorrect values were printed when individual fields were accessed.

The Debugger treated a packed array of unsigned bytes as if the array elements were signed, causing elements larger than 127 to print as negative.

Debugger, Real Constants

The Debugger gave incorrect results in writing the value of real constants.

/FTN Switch

When files were opened with the `/ftn` switch, the second character on the line was being used to position the cursor on CRTs. The fix moves the carriage-control character to a register to be sure that the upper byte is zero. A solution for users with previous versions is to write a null character (`chr(0)`) as the second character on the output line, immediately following the FORTRAN control character.

Text Files and 'Put'

Creation of a text file with `put` operations, rather than `write` or `writeln`, altered one extra byte past the end of the allocated buffer. This would zap whatever data happened to be on the heap after the file buffer. The overflow was detected, but the heap was damaged.

Pascal-1

V1.3 bug fixes are currently in progress. A summary of the fixes will be included with the update release. Two fixed thus far are:

Abs() problem

The **abs()** function gave incorrect results for **real** operations. It now gives correct results for both reals and integers.

Trapping on underbars

A program containing underbars in identifier names caused the compiler to trap. It now gives an error message.

Pascal
NEWSLETTER
OREGON SOFTWARE

2340 SW Canyon Road
Portland, Oregon 97201

Collins Hemingway, editor
David Spencer and Michael Kuhn, staff writers

The Pascal Newsletter is published quarterly by Oregon Software, Inc., 2340 SW Canyon Rd., Portland, OR 97201; (503) 226-7760. Each customer of Oregon Software receives one free subscription per site. Additional subscriptions are available upon written request.

The Pascal Newsletter accepts articles of interest to Pascal users: solutions to troublesome programming situations, new applications of Pascal, interesting variations on standard applications, etc. Submit articles on paper (typed and double-spaced), on floppy disk, or on magnetic tape, sent to the attention of the editor. We pay $100 per newsletter page for any article we print.

Information exchange

If you need information on technical applications involving Pascal, or if you have an application that might interest other users, send us a brief description for inclusion in the Information Exchange. Your description should follow the format of the items below. Interested parties can contact one another directly.

Data Base written in standard Pascal. Intended as an educational tool and currently used by computer science classes, the working system consists of four interactive programs, with a screen updating program and a calculation routine scheduled for inclusion this summer. For information, contact: Leon Schilmoeller, Computer Science Dept, Augustana College, Souix Falls, SD 57102, (605) 336-5495.

Pascal Users' Group. The new contact person for the PUG is: Charles Gaffney, 2903 Huntington Rd., Cleveland, Ohio 44120.

Pascal Programmer sought by dynamic, research-oriented company. Ideal candidate should have a thorough knowledge of Pascal programming and be familiar with PDP-11 minicomputers and/or the Motorola MC68000 microprocessor. Send letter of inquiry and resumé to: Personnel Director, P.O. Box 1201, Portland OR 97207.

Compiler Writer wanted. Dynamic, research-oriented company seeks senior software engineer with a thorough knowledge of Pascal programming and an extensive background in writing code generators. Send letter of inquiry and resumé to: Personnel Director, P.O. Box 1201, Portland OR 97207.

OPUS Communiqué

Oregon Pascal Users Society

TUSTIN, California — As of February, Oregon Pascal User's Society (OPUS) has more than 100 members. At the first two OPUS meetings held in Irvine, California, the 10 attending members of the group decided that the organization should be structured as two major subdivisions: United States and International.

Within the United States, OPUS will be divided into subgroups by time zone, e.g., West Coast, Mountain, Central, and East Coast. The International OPUS will be subdivided by geographical proximity. The international group is not yet subdivided because of the small number of international members (so far) and their extreme geographic diversity. Canada, with 10 to 15 members, is an exception.

Many local OPUS (LOPUS) chapters can be formed within each time zone. Now, we need people in the local areas who will accept the responsibility of organizing a local OPUS. The Southern California OPUS (SCOPUS) is an example of the local OPUS structure that most members felt would be useful in attacking the problems of certain tasks they have encountered. The Southern California chapter held its first two meetings in January and February 1983.

Anyone interested in starting a LOPUS should contact Bruce Williams, the OPUS coordinator, at the address given below. He will help you get started.

OPUS Projects

OPUS is conducting a survey to determine the machines and compilers currently in use by its members and the tasks (business applications, systems, etc.) for which those systems are used. The questionnaire is being mailed to each OPUS member. Responses will be collected in March and April. After compiling the results, OPUS will send to each member a membership list and a copy of the results. With the results of the survey, an OPUS member can find out who in their area is using Oregon Software's Pascal and in what applications. If you aren't a member or if you have not received a member survey, please write to Bruce Bruce at OPUS.

We're also putting together a library of useful routines submitted by OPUS members.

Notes from OPUS Members

From EOCOM in Southern California — Recently, Lyle Norton discovered that the **time** function for RT-11 takes care of the 50/60 Hz conversion before it returns the time to the calling routine. You don't have to add your own conversion code. Isn't it nice to have things done right *before* you start the coding? You won't, however, find this information in the manual.

From Allergan in Southern California — Jerry Shaver has done some nice routines using character I/O on RSX-11M. Jerry said he would prepare them for the OPUS library being developed. If you're curious about these I/O routines, contact him through OPUS.

Next Communiqué

We'll say more about the OPUS library of routines and give you some guidelines for writing code that you want to submit to the library. If you want to appeal for a solution to some nagging problem or the ways others may have attempted to do something, please write to OPUS at the address below. We'll get your appeal into the Communiqué.

Bruce Williams,
OPUS Coordinator
c/o EOCOM
15771 Red Hill Ave.
Tustin, CA 92680
(714) 730-5051, ext. 302

Editor's Note:

The User's Manual for RT-11 says that the built-in function `Time` "returns a real value corresponding to the current time of day" on page 57. We saw no need to explain in detail how the time was produced.

Anne Smith retires; Pat Rau named manager

Anne Smith, Manager for General Distributors, retired on March 31. As a founder of Oregon Software, she has spent the last seven years helping the company grow and succeed, building up our distributor network and creating the sales group.

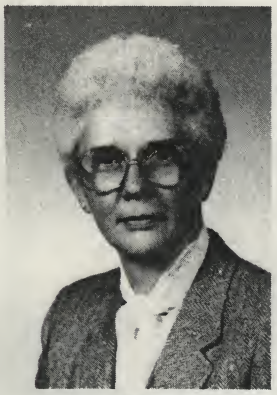
Retirement is probably the wrong word for Anne's future; redirection of her energy would be better. As Anne says, "There is so much to do that I always feel I need to rush out and do it all TODAY! That's one of the reasons for early retirement — to have the time to do it."

Travel is first on her new priority list. Anne plans a two-week cruise through the Caribbean islands, including two stops in South America. "I'm really going to tourist it," she says. Of course, she'll be doing a **lot** of fly-fishing on our beautiful western trout streams, and she'll be cross-country skiing in the winter, mid-week when the trails are not so crowded. She also wants to take a two-week raft trip on the Colorado River, through the Grand Canyon this year or next.

Anne wants to resume volunteer work for environmental groups such as the Nature Conservancy and Audubon Society. She used to be active in these groups, but hasn't had time during her years with Oregon Software. Her close friend Pete Pedersen, a lawyer, works as a volunteer for the ACLU and she may find something there, too.

From time to time, Anne will serve as a consultant to Oregon Software and she may fill in for vacationing sales personnel occasionally.

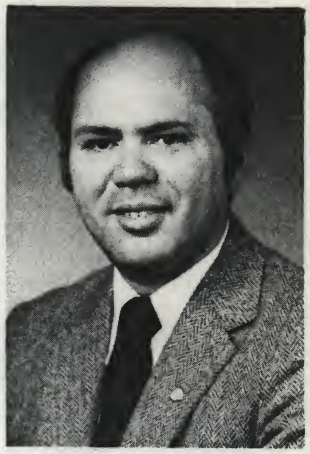
Pat Rau succeeds Anne as Manager for General Distributors. Pat has been with Oregon Software for two and a half years, as a general salesperson and specialist in licensing agreements. Before joining Oregon Software, Pat worked for Honeywell's Portland division.



Anne Smith

Pat Rau

David Cloutier heads marketing



David Cloutier

Oregon Software has appointed David Cloutier, formerly a software marketing manager at Intel, as vice president of marketing and sales.

David was the software marketing manager of OEM microcomputer systems at Intel's Hillsboro, Ore., division, and was product manager for the introduction of Intel's iRMX 86 operating system. Additionally, he served as chairman of the OEM division's software business planning committee. All together, David has 10 years of software marketing experience, having also worked at Zentec and Perkin-Elmer.

David expects Oregon Software to become a major software house for the MC68000, based on recent releases of a family of software products for developing high-performance applications for Motorola's MC68000. David also expects the company to continue its enhancement of products serving the DEC market, both through major upgrading of compiler products, such as the 2.1 release, and through new products such as SourceTools.

"The professional programmer is the primary focus of our work," David said. "That's because we are a company of professional programmers. Every tool we sell is a tool we first develop for our own use, and every tool we sell has to first meet our own high expectations. We must bring that quality advantage to bear in the market to give our customers the same high performance in their applications."

User manuals revised for V2.1/1.3

New editions of all PDP-11 manuals for Pascal–2 are expected to be published in late spring. New editions of Pascal–1 manuals will be printed in the fall.

The new editions will document all new software features included with the releases of Versions 2.1 and 1.3, respectively. The Pascal–2 manuals will include new sections, expanded sections, additional examples in many areas, corrections of errors and omissions, and inclusion of all current errata. The manuals also will have indexes.

Pascal–1 manuals will be upgraded to include many of the features now found in the Pascal–2 manuals, including tutorial sections and index.

Pending the printing of the new editions, the new 2.1/1.3 features will be documented in a supplement to the current manuals. The supplement will be shipped with all updates.

User manuals for all Oregon Software products will take a new form with all new printings: looseleaf with windowed covers and binders. (You remember the binders, don't you?)

This format will allow us to send send future errata as inserts, in the form of "change pages." Users can "mix and match" their Oregon Software library according to their choice of compiler and options. The manuals will now lay flat during use. The new cardboard cover allows users who preferred the bound editions to keep a semblance of that feel.

Book Prices

As of April 1, 1983, new prices will be in effect for all books and manuals available from Oregon Software. The new price list follows:

User Manual	Type of Sale	Price
Pascal-1 and Pascal-2	End User	$15.00
	Volume Purchases/Instructional	$10.00
SORT-1-Plus	End User	$10.00
	Volume Purchases/Instructional	$ 7.00
SourceTools	End User	$10.00
	Volume Purchases/Instructional	$ 7.00
Concurrent Programming Package	End User	$15.00
	Volume Purchases/Instructional	$10.00

Book	Author(s)	Price
Algorithms + Data Structures = Programs	Wirth	$27.95
Elements of Programming Style	Kernigan, Plauger	$14.95
Introduction to Pascal	Zaks	$15.95
Pascal User Manual and Report	Jensen, Wirth	$ 9.50
Programming in Pascal	Grogono	$18.95
Structured Programing	Dahl, Dijkstra, Hoare	$20.00
Systematic Programming: An Introduction	Wirth	$26.00
Tex and Metafont	Knuth	$12.00
Concurrent Euclid, The UNIX System, and Tunis	Holt, Graham, Lazowska, Scott	$15.95

Books and manuals are shipped FOB destination (within USA) or FOB Portland (outside USA).

2340 SW Canyon Road, Portland, Oregon 97201 Printed in USA